DEVELOPING A FRAMEWORK FOR A NEW VISUAL-BASED INTERFACE DESIGN IN AUTODESK MAYA

A Thesis

by

TIMOTHY CLAYTON WITHERS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2012

Major Subject: Visualization

DEVELOPING A FRAMEWORK FOR A NEW VISUAL-BASED INTERFACE DESIGN IN AUTODESK MAYA

A Thesis

by

TIMOTHY CLAYTON WITHERS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Co-Chairs of Committee,	Joshua Bienko
	Philip Galanter
Committee Members,	Tracy Hammond
	Justin Israel
Head of Department,	Tim McLaughlin

August 2012

Major Subject: Visualization

ABSTRACT

Developing a Framework for a New Visual-Based Interface Design in Autodesk Maya. (August 2012)

Timothy Clayton Withers, B.S., Texas A&M University

Co–Chairs of Advisory Committee: Joshua Bienko
Philip Galanter

In this thesis, I develop an efficient and user-friendly node-based interface to be used in the creation of a particle system in Autodesk Maya. Maya's interface inconsistencies were identified and solutions were designed based on research in a number of fields related to human-computer interaction (HCI) as well as taking design queues from other highly successful 3D programs that employ a node-based interface. This research was used to guide the design of the interface in terms of organizing the data into logical chunks of information, using color to help the user develop working mental models of the system, and also using simple, easy to identify, graphical representations of a particle system. The result is an easy-to-use and intuitive interface that uses a visual-based approach in creating a particle system in Maya. By following guidelines laid out by previous researchers in the field of HCI, the interface should be a less frustrating to use and more organized version of Maya's current interface.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

The goal of this work is to create an efficient and user-friendly node-based interface to be used in the creation of a particle system in Autodesk Maya. Maya was chosen because of its outdated interface as well as it being one of the very few 3D programs that does not use a node-based approach to create particles. Maya's interface inconsistencies were identified and solutions were designed based on research in a number of fields related to human-computer interaction (HCI) as well as taking design queues from other highly successful 3D programs that employ a node-based interface. This research was used to guide the new design of the different interface aspects. I discuss this research in depth and indicate how I specifically used it to direct the design of my interface. The result is an easy-to-use and intuitive interface that uses a more modern-based approach in creating a particle system in Maya.

I.1. Motivation

Human-computer interaction is an interdisciplinary research area, which concerns the study of interaction between humans (operators as users) and computers. Its ultimate goal is to contribute to improving the interaction between humans and computer systems by endowing technical systems with higher usability and satisfaction [39].

In HCI, real users are represented by a term called "User profiles." The user profile is a method of presenting data from studies of user characteristics [14]. The ultimate purpose of using user profiles or persona is to help designers to recognize

---

The journal model is *IEEE Transactions on Visualization and Computer Graphics.*

or learn about the real user by presenting them with a description of a real user's attributes. For instance; the user's gender, age, educational level, attitude, technical needs and skill level. User profile does not necessarily mirror or present a complete collection of a whole user population's attributes. The essence of user profiles is an accurate and simple collection of user's attributes [39]. Given the amount of variables that are present in designing an interface, it is understandable that creating an intuitive GUI that is easily used by a wide variety of users is a difficult task.

It has been estimated that 48% of the work on a computer project goes into the design and implementation of the user interface. Recently, Douglas, Tremaine, Leventhal, Wills, and Manaris (2002) noted the importance of Human-computer interaction (HCI) suggesting at least 50% of programming code is devoted to the user interface [28].

The most widely used interface type in computer applications is the (W)indows, (I)cons, (M)enus, and (P)ointing devices, or the WIMP interface. Psychologically, WIMP interfaces provide the advantage of allowing the user to visualize what is going on, recognize operations instead of having to recall commands (recall is generally much more difficult than recognition), transfer knowledge about the architecture of the physical interface from one application to another, and achieve high compatibility between stimulus and response [7]. However, in many complex computer graphics applications with hundreds and thousands of different unique commands such as Autodesk Maya, defining a clear and consistent system of graphical icons can be an overwhelming task and can often result in more confusion for the users [22]. Furthermore, displaying each of the, potentially, thousands of commands can become cumbersome and overwhelming.

A majority, if not all 3D programs, such as Maya, SideFX's Houdini, Exotic Matter's Naiad, and the Foundry's Nuke, are built on the WIMP interface model;

each have menus that display different options that the user can modify. The bulk of the modifiable options come from objects (polygon surfaces, NURBS surfaces, particle systems, etc) that the user creates in the 3D scene. Connections can be made and relationships established between objects so that one affects the other. It is important to see these connections when working on a project. The biggest difference between Houdini (or Naiad) and Maya is how users create objects and connections.

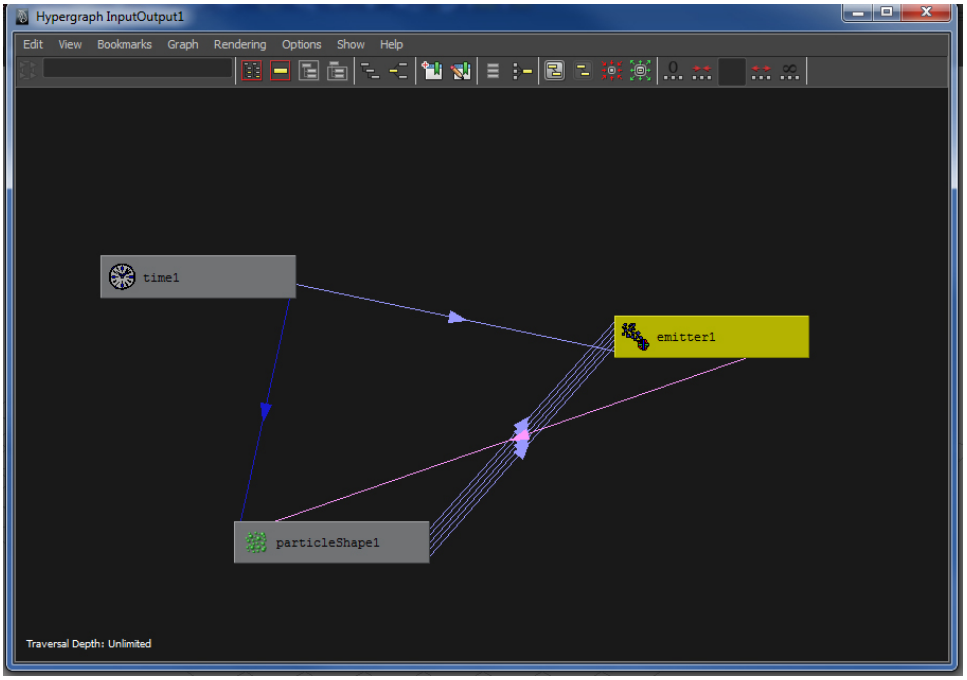


Fig. 1. Maya 2011 Dependency Graph.

Maya users create objects and connections through toolbar menus. Nodes (icon-based graphics) and connections are then created automatically and are free to be viewed and modified in a window called the hypergraph. The hypergraph shows hierarchies and dependencies of objects. Individual attributes of a node are connected to other nodes and are represented by a line and an arrow. However, understanding

the connections between objects in the hypergraph can be extremely difficult because there are often multiple connecting lines of varying colors between nodes with obscure names.

Fig. 1 shows a very simple particle system within the dependency graph window (also called the Hypergraph). Thus, interaction with the object nodes and their connections is rare and aimed at only advanced users that can understand the variables associated with the node connections.

In contrast, Houdini and Naiad's entire interfaces are based around user created nodes within a node graph window. The node graph window is a simplified version of Maya's hypergraph and houses all nodes in a scene. These nodes not only represent 3D objects in the scene, they also contain menus with different options the user can modify. Connections are made, not automatically, but by users manually dragging a line between nodes. This allows users to easily see and understand how objects are affecting other objects in the scene.

The benefit of designing a GUI around user created nodes is that it is designed for organization, efficiency, and to minimize and handle complexity. Generally, each node can either receive and/or send data to and from other nodes that are connected to it. The node then modifies that data and sends it on to the next node. This system offers the flexibility to interchange nodes without the fear of a scene irreparably breaking. This interchangeability allows for rapid prototyping of a wide-variety of different options with a few simple clicks, rather than setting up an entirely new scene all together.

I have chosen to revamp a portion of Maya's GUI by developing and designing a more modern node-based interface that will assist users in creating a particle system. I am focusing on Maya's interface because it is one of the most, if not the most utilized 3D program in the visual effects industry today and the industry, as a whole, seems

to be shifting in the direction of node-based interfaces. I am focusing specifically on particle systems because particles are the driving force for most effects in games, movies, and advertising. However, the framework that I have put in place with this interface is a good foundation for someone to expand into other areas of Maya.

CHAPTER II

RELATED WORK

II.1. Graphical User Interfaces

II.1.1. History

Graphical user interfaces did not really exist until the late 1960's. Before then, computers were giant mainframes. Typically, users would interact with them using what was called "batch processing." A user would submit a program on a series of punch cards, the computer would run the program at some scheduled time, and then the results would be picked up hours or even days later [29]. The idea of having users enter commands on a text-based terminal in real-time was considered radical [29]. That all changed with Douglas Englebart, an employee at the Stanford Research Institute and considered to be the "father" of the GUI.

Douglas Englebart and his staff worked for years to develop the first working GUI called the NLS, or oN-Line System–named for its ability to network between multiple computers. The NLS could display both text and solid lines and was based on vector graphics technology. Douglas' hands operated three input devices: a standard typewriter-style keyboard, a five-key "chording keyboard" (combinations of the five keys could produce 32 separate inputs, enough for all the letters of the alphabet), and a small rectangular box about the size of a couple of juice boxes with three buttons near the top, connected to the computer with a long wire [29]. This box with three buttons was dubbed the "mouse." With the invention of the mouse came the invention of the mouse pointer, which let users know what part of the interface they were affecting with the mouse. The NLS featured hypertext linking, full-screen

document editing, context-sensitive help, networked document collaboration, e-mail, instant messenging, and video conferencing [29].

The NLS was a revolutionary product and research and the institute continued until lack of funding caused it to close down in 1989. However, the NLS opened the door to the Alto, a computer developed by Xerox PARC. The Alto was not a microcomputer, even though its working components did fit in a minibar-sized tower that fit under a desk. Its most striking feature was its display, which was the same size and orientation as a printed page, and featured full raster-based, bitmapped graphics at a resolution of 606 by 808 [29]. Unlike the vector-based NLS that could only display text and straight lines, the Alto's display had the ability to turn on and off each pixel independently.

The first few programs developed for the Alto were only slightly graphical and rather crude. The biggest problem between all of the programs was the fact they did not have a consistent user interface. Luckily, the PARC researchers realized this flaw and developed a new visual code development environment called Smalltalk, shown in Fig. 2, the first modern GUI.

What was amazing about Smalltalk wasn't that it was the world's first object-oriented programming language and had modern, Java-like features like automatic memory management, it was its graphical development environment that introduced many modern GUI concepts. Smalltalk introduced windows that had title bars and that could lay on top of each other. Small iconic representations of programs or documents, called "icons," were invented so that users could click and launch whatever program was associated with it. Popup menus, scroll bars, radio buttons, and dialog boxes were also invented. The combination of Smalltalk and the Alto was essentially a modern, albeit expensive, personal computer.

After the invention of Smalltalk and its GUI elements, computer companies tried

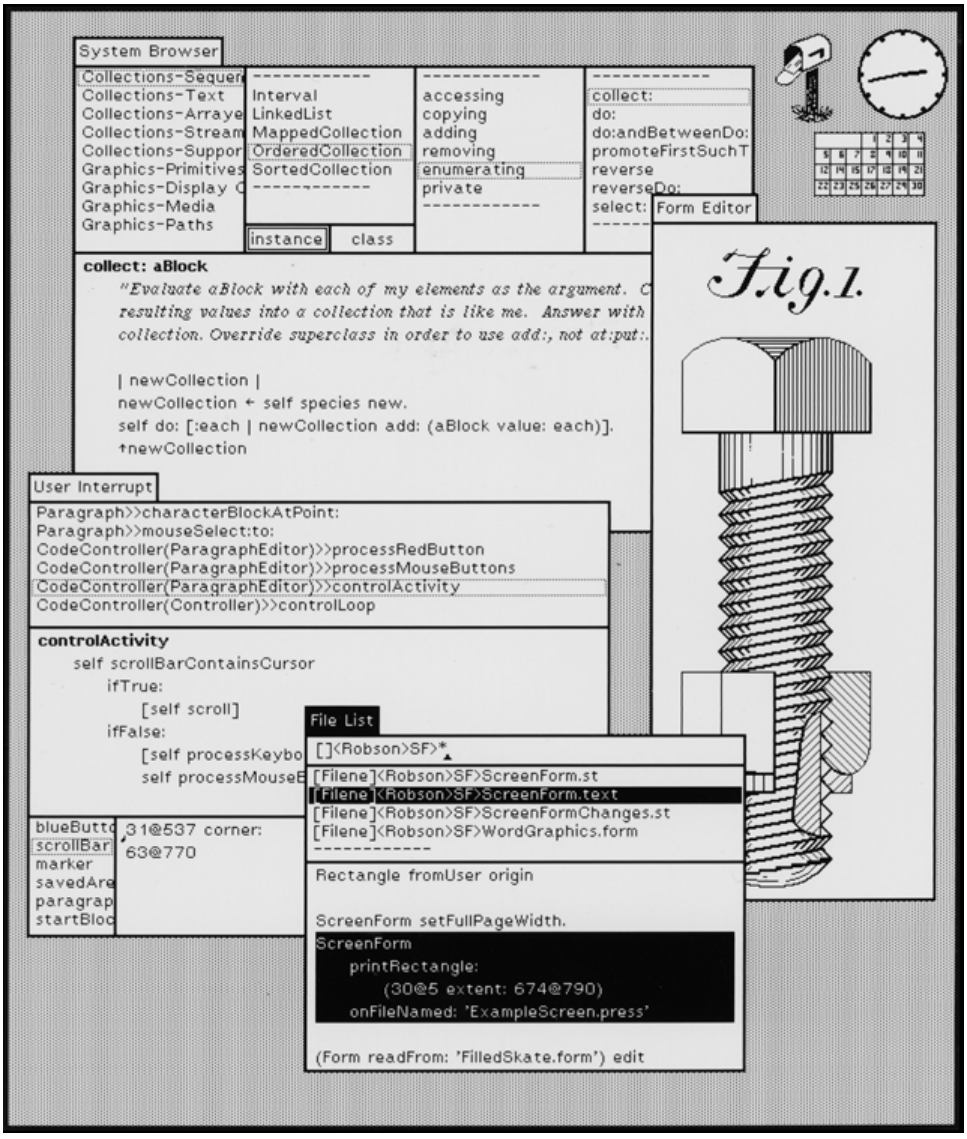


Fig. 2. The Smalltalk Development Environment.

to extend the graphical user interface and make it their own. VisiCorp developed a clunky failure called VisiOn. Tandy Computers released a GUI of their own named DeskMate that was hard to use and did not achieve much success. Other companies would create GUIs such as the Amiga, GEM, GEOS, and ACORN. However, Apple, with its willingness to take risks and huge cash flow, would be the most important of the GUI pioneers. Microsoft too would go on to be an integral aspect of GUI design and would develop the most popular GUI of all time with the release of Windows 95.

II.1.2. Elements of a Graphical User Interface

There are many aspects to a user interface but in the following section I will briefly summarize what I feel are the two most important characteristics for users and programmers to be aware of. One is concerned with the look and the functionality of the GUI, called "widgets," and the other is how the interface identifies and responds to events, or actions, taken by the user.

II.1.2.1. Widgets

In computer programming, a widget is an element of a graphical user interface that displays an information arrangement changeable by the user. GUI widgets span a large range of interaction solutions: selecting a file; triggering an action; choosing options; or even direct manipulation of graphical objects [24]. The defining characteristic of a widget is to provide a single interaction point for the direct manipulation of a given kind of data. Widgets are basic visual building blocks, which, combined in an application, hold all the data processed by the application and the available interactions on this data. Basic widgets include, and are not limited to, buttons, labels, text editors, scroll bars, progress bars, radio buttons, and check boxes. Custom

widgets can be built by the programmer and be made up of multiple widgets with custom or default functionality. For example, a custom widget could include three checkbox widgets, two button widgets, and a text editor widget.

II.1.2.2. Signals

Every GUI library has a way of providing details of when external events take place, usually triggered by a user, such as button presses, keyboard presses, or mouse clicks. These events are then processed by something called a "callback" or "event handlers". These are usually a function or a method in the program that is executed after an event occurs. Depending on the GUI library, the programmer may also have access to a wealth of other information when an event is triggered; it is possible to know the coordinates of the mouse click relative to the widget and relative to the screen, the state of the Shift, Ctrl, Alt, and NumLock keys at the time of the click, the precise time of the click and of the release, and so on [37].

II.1.3. Considerations in the Design

The human system, as with all other living systems, is susceptible to information overload; at some point the inputs entering the system become so great that they can no longer be processed effectively [21]. However, it has been found that the human information processing system can handle considerably more inputs if those inputs are received on multiple channels [13]. The information capacity of a single information channel is approximately seven [20]. There is, however, considerable variation between the various channels that are commonly employed (for instance– line thickness, color, brightness, symbol shape). Color, for example, can transmit considerably more information in a coding scheme than can line thickness or display

brightness [13].

Unfortunately there are no concrete set of rules for designing a great looking and functional GUI. There have however, been several general principles or guidelines that have been accumulated through various research studies on the human visual perception system and graphic design. The guidelines most pertinent to this thesis can be grouped into two different categories: Organizing the Data and using Color to aide the user.

II.1.3.1. Organizing Data

In 1982, H.E. Dunsmore showed that screen clarity and readability could be improved by making screens less crowded. By splitting items on one long line into two, he showed, improved productivity by 20%. In 1983, R.S. Kesiter and G.R. Gallaway showed that "good design principles" improved productivity 25% while reducing errors by 25% [32]. But why does this improve productivity? One reason could be pure physics. D. Sarna and G. Febish [32] report that at a distance of 2.5 degrees from the point of eye fixation, visual acuity (the ability of the eye to resolve detail) goes down by half. At a normal viewing distance of about 19 inches, this translates into a circle with a diameter of 1.7 inches. In other words, if someone focuses at a point on the screen, what's in sharp focus is a circle around that point with a diameter of 1.7 inches. This constitutes a single chunk of information that the eye takes in at a glance. This is called the rule of 1.7. Sarna and Febish go on to say that if a person needs to constantly move his or her eyes across the screen, they are forcing a lot of unnecessary and tiring eye movement in order to read each chunk, and the rule of 1.7 is being violated. Their company, ObjectSoft, has three general principles of good GUI design as it pertains to organizing information: (1) Arrange the screen into logical chunks of data (one task, one chunk); (2) Group chunks within frames;

and (3) Line the chunks up neatly in rows or columns or squares.

In 1980, A. Marcus [17] discussed six principles that describe human perception as they relate to computer graphics. Two of these six principles are, what I feel, to be the most important as it relates to my interface design: Proximity - Objects that are located near each other will appear to belong together. For example, two words appearing next to each other on a graph will be perceived as a common label; and Similarity - Objects that share a similar characteristic (e.g., color, size, shape) will be perceived as belonging together. Color, for example can be used to tie together two items on a graph that do not otherwise seem to be related.

II.1.3.2. Color as a Design Aide

People interact with the world around them through the mental models they have developed [38]. Specifically, the ideas and the abilities they bring to the job are based on the mental models that they develop about that job [26]. An interface needs to help the user develop an effective mental model of the system. Color can help develop workable, efficient mental models because the proper use of color can communicate facts and ideas quickly to the user. Color can be a powerful tool to improve the usefulness of an information display in a wide variety of areas if color is used properly. Conversely, the inappropriate use of color can seriously reduce the functionality of a display system [23]. Information overload is a likely consequence if a designer overuses color or the colors do not correspond with the user's prior color expectations [8].

A designer should adhere to a small color palette. Overusing color makes the display look cluttered and can confuse users making their tasks more complex, increasing errors, and reducing their productivity [16]. Robertson [30] notes that users frequently give negative feedback for screens designed with more than four colors.

Durrett and Trezona [11] suggest using no more than four colors with novice users and a maximum of seven if designed for experienced long term users. Furthermore, Murch [23] says that the number of colors in an interface should not exceed five plus or minus two. According to Miller [20] the magic number for short term memory is seven plus or minus two. If the user is overwhelmed or confused by too many colors vying for his or her attention, he or she is unlikely to develop an effective mental model [18].

The types of colors used for an interface also can affect efficiency of use. Roughly 64% of the cones in the human eye contain red photopigments, 32% contain green, and only about 2% contain blue photopigments [23]. Due to the lack of blue photoreceptors, thin blue lines or small blue objects tend to blur or disappear when someone tries to focus on them. This does not mean to suggest to never use blue in an interface, it merely suggests that the designer should be careful with the use of the color blue. Also, if colors are to be overlaid on or near one another, colors with good contrast to each other should be used. For example, white text on a black background will be easier to read than yellow text on a white background.

The intuitive ordering of colors can help establish consistency in the interface design, which is one of five "golden rules" MacDonald [16] talks about when using color effectively in computer graphics. Color can be used to encode or chunk information items. This helps increase the number of items a user can retain in short-term memory [38]. Experiments have shown that the search time for finding an item is decreased if the color of the item is known ahead of time, and if the color only applies to that item [27]. That means that the clear use of color will aide the user in finding what he or she wants to quickly and efficiently. Additionally, usability of the interface can be increased by assigning colors to chunk information about subsystems and structures [38]. In fact, errors in understanding and using the interface can actually be reduced

by using color to clarify the system meanings and concepts [18].

II.2. Visual Programming

Aaron Sloman distinguishes two kinds of data representations: "analogical" and "Fregean" (named after Gottlob Frege, the inventor of the predicate calculus). Analogical representations are similar in structure to the things they describe; Fregean representations have no such similarity [35]. See Fig. 3. Traditional programming can be compared with "Fregean" data representations and visual programming can be compared with "analogical" data representations.

Traditional programming can be defined as specifying a method for doing something a computer can do in terms the computer can interpret [34]. A person programs by writing computer instructions in a programming language, compiling that code, and then executing it. This used to be a daunting task because computer languages were originally designed with efficiency in mind and not if the language was easy to write as a programmer.

According to Brad Myers, a professor of Human-Computer Interaction at Carnegie Mellon University, visual programming refers to any system that allows the user to specify a program in a two (or more) dimensional fashion [25]. In terms of a broader sense, Nan Shu defines visual programming as the use of meaningful graphical representations in the process of programming [5].

Some of the most important names in the field of visual programming, including Chang, Shu, and Burnett, worked on identifying the defining characteristics of the major categories of visual programming languages [6], [5]. They are: (1) Purely visual languages; (2) Hybrid text and visual systems; (3) Programming-by-example systems; (4) Constraint-oriented systems; and (5) Form-based systems.

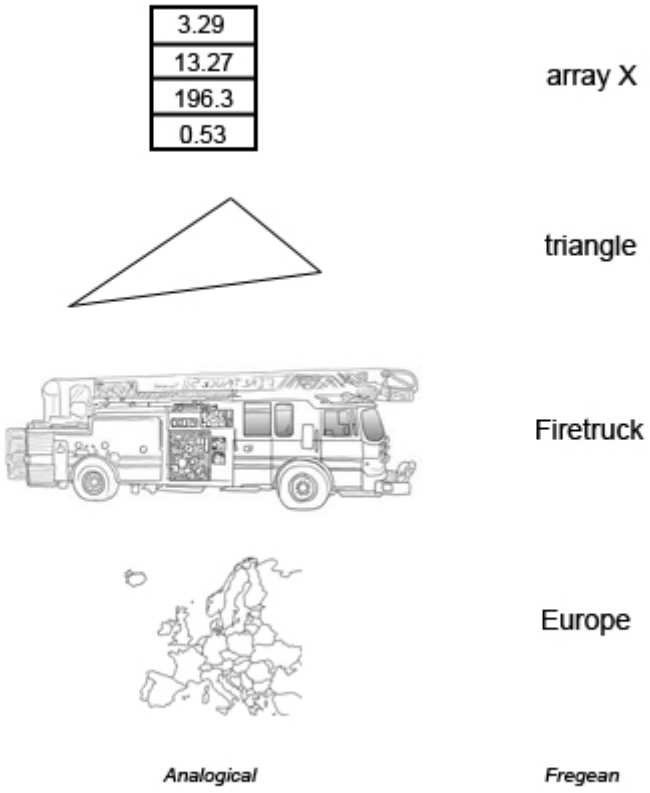


Fig. 3. Analogical vs. Fregean Representations of Data.

The implementation of this thesis focused on the hybrid text and visual systems characteristic. This hybrid system includes both those in which programs are created visually and then translated into an underlying high-level textual language and systems which involve the use of graphical elements in an otherwise textual language [4]. In comparison, my interface creates a particle system with 2D icon graphics and allows the modification of attributes with graphical widgets. The attributes that are changed are then translated into a function that Maya can execute. This system

makes it so that users only need to know how to interact with the nodes themselves and do not need to know the specifics of how the node's underlying algorithms are being executed.

II.3. Why Visual Programming?

Many people relate to the world in an inherently graphical way and use imagery as a primary component of creative thought. It is suggested that it can be easier for a person to identify and create a mental relationship with a pictorial representation of a subject (analogical representation) rather than a textual representation (Fregean representation). Jacques Hadamard, in a survey of mathematicians found that many mathematicians and physicists think visually and reduce their thoughts to words only reluctantly to communicate with other people [12]. Even Albert Einstein said, "The words of the language, as they are written or spoken, do not seem to play any role in my mechanism of thought." [36].

Jerome Bruner identified three ways of thinking when it comes to learning: (1) Enactive - in which learning is accomplished by doing; (2) Iconic in which learning and thinking utilize pictures; and (3) Symbolic in which learning and thinking are by means of Fregean symbols. This is the main goal of education today.

However, Bruner argues that it is a mistake for schools to abandon enactive and iconic thinking skills when learning the symbolic. All three mentalities are valuable at different times and can often be combined to solve problems. All three skills should be preserved [36].

Visual programming helps a person reduce or remove entirely the necessity of translating visual ideas into somewhat artificial textual representations [4]. It is made popular by the notion that: (1) pictures can convey more meaning in a more

concise unit of expression; (2) pictures can help understanding and remembering; (3) pictures can make programming more interesting; and (4) when properly designed, pictures can be understood by people regardless of what language they speak [34].

Pablo Picasso once said, "The most awful thing for a painter is the white canvas." A blank coding pad is as much a barrier to programming creativity as a blank canvas is to a painter [36]. Pygmalion was developed with the intention of being a non-threatening medium that a creative person could use as a starting point for their programming ideas.

II.4. GUI Toolkits

It is taken for granted that GUI application designers and programmers can reuse existing interaction solutions embodied in GUI toolkits and widget libraries [31]. GUI toolkits have been created for interface designers to make maintaining and developing interfaces much easier. According to Salber, Dey, and Abowd [31] there are three main benefits to GUI toolkits: (1) They hide specifics of physical interaction devices from the applications programmer so that those devices can change with minimal impact on applications. Whether the user points and clicks with a mouse or fingers and taps on a touchpad or uses keyboard shortcuts doesn't require any changes to the application; (2) They manage the details of the interaction to provide applications with relevant results of user actions. Widget-specific dialogue is handled by the widget itself, and the application often only needs to implement a single callback to be notified of the result of an interaction sequence; and (3) They provide reusable building blocks of presentation to be defined once and reused, combined, and/or tailored for use in many applications. Widgets provide encapsulation of appearance and behavior. The programmer doesn't need to know the inner workings of a widget to use it.

II.4.1. GTK+

GTK+ (GNU Image Manipulation Program Toolkit, or GIMP Toolkit) was started as a toolkit for the GIMP around 1996 and reached its first stable release in April 1998 [9]. It was originally created by Spencer Kimball, Peter Mattis, and Josh MacDonald as the GUI library for the open source GNU Image Manipulation Program (Linux's version of Adobe Photoshop) [22]. GTK is cross platform and open source, but it requires X11 libraries to run on OS/X which does not allow programmers to exploit the unique GUI features of OS/X as is possible with Qt, another GUI toolkit [10].

II.4.2. FLTK

FLTK (pronounced "fulltick") stands for the Fast Light Tool Kit and is a LGPL'd C++ graphical user interface toolkit for X (UNIX), OpenGL, and Windows. Bill Spitzak first developed it because the company he was working for at the time, Digital Domain, needed to switch to OpenGL and GLX and wanted to use C++ subclassing. FLTK has a very low learning curve and is ideal for beginners to interface programming. However, because of the ease-of-use, FLTK is not as full-featured and does not offer the extensive customizability of some of its competitors.

II.4.3. Qt

Qt was initially developed by Haavard Nord and Eirik Chambe-Eng in 1991 and was first publicly available in 1995. Key features of the Qt library includes platform independence and a meta object compiler, which allows for the use of programming concepts not present in C++ such as introspection (runtime determination of object types) and a signal/slot system–invented by Eirik Chambe-Eng–to allow for asynchronous input and output and event driven programming constructs [22]. Addi-

tionally, Qt also has a full 2D graphics API with support for rotations, scaling, and shearing. Qt is used for its highly customizable widgets, its wide variety of widgets, and because it is available through many different programming languages: Python, Ruby, C#, Java, etc...

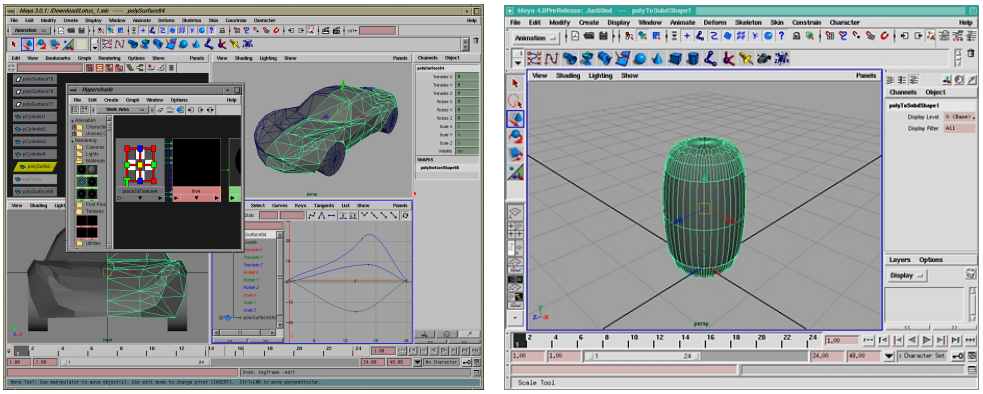
II.5. Existing Node-Based Interfaces

II.5.1. Autodesk Maya

II.5.1.1. The Interface

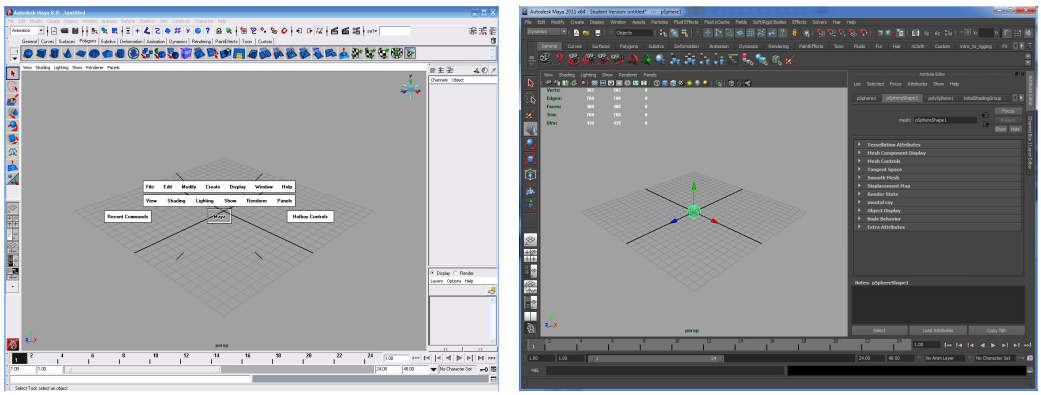
Autodesk Maya is in its 12th version. Although significant advancements have been made in Maya's fluids, animation, modeling, and rendering, the user interface has changed very little. See Fig. 4 and Fig. 5. Up until Maya 2011, the interface for has been built mainly with Maya's own scripting language called Maya Embedded Language, or MEL. Not only was this scripting language used to build Maya's interface, it also was used for most of Maya's core functionality. Additionally, MEL is a means for users to customize the functionality and the look of the software and also offers the ability for users to make interfaces for use within Maya. However, depending on the project, this would require advanced scripting experience and MEL offers only basic widgets when it comes to interface design and is not very flexible.

In addition to MEL, Python is also been fully integrated into Maya since version 8.5. With the development of PyCmds, PyMEL, and PySide, a large majority, if not all, of Maya's functions can be executed with Python script. The only drawback to using Python instead of MEL is the fact that Python commands are wrapped around MEL commands. This means that when a Python command is run, it has to be converted into a MEL command before it is executed. However, depending on the project, the time taken to do this is relatively small and unnoticed by the user.



(a) Maya 3.0, 2000 (b) Maya 4.0, 2001

Fig. 4. Examples of Maya's Past Interfaces: (a) Alias Wavefront Maya 3.0, Released in Feb. 2000. (b) Alias Wavefront Maya 4.0, Released in June 2001.



(a) Maya 8.0, 2006 (b) Maya 2011, 2010

Fig. 5. Examples of Maya's Past Interfaces: (a) Autodesk Maya 8.0, Released in Aug. 2006. (b) Autodesk Maya 2011, Released in April 2010.

II.5.1.2. The Dependency Graph

The controlling structure inside of Maya is the dependency graph (DG). The DG is a collection of nodes of objects that have been created in the scene. Each node in the dependency graph represents an action to build up or change the scene. The final result is the scene in its current state. A node takes in a set of input data (supplied by connections to other nodes) and uses it to create a set of output data. The data that flows through the nodes can be as simple as numbers or as complicated as a surface.

II.5.2. SideFX Houdini

Houdini is a visual dataflow modeling and animation tool. Visual Dataflow Modeling (VDM) is a procedural approach to creating design models [15]. It allows designers to efficiently explore alternative forms without having to manually build each different version of the design model for each scenario. Such systems are used by architects and engineers to automate design exploration and accelerate the design process [3]. Houdini not only uses this approach for modeling, but also for animation, effects, rendering, and compositing.

In contrast to Maya, Houdini's entire workflow is based around users creating and linking nodes together. This node-based interface can be thought of as a type of visual programming language that users will learn as they get more familiar with the program. Nodes can be added, deleted, bypassed, or commented on without the fear of a scene irreparably breaking, which could force the user to start over. By basing Houdini's workflow around nodes, all the steps needed to setup a shot are represented by the node networks. The advantage of this is that information can be retained deep into production and can be used to make last minute creative decisions that would

be too costly in a traditional CG pipeline [1].

II.5.3. Naiad

Exotic Matter was founded in 2008 by Marcus Nordenstam and Dr. Robert Bridson [33]. Naiad is a relatively new dynamics solver and simulation framework capable of producing animations that consist of liquids, breaking waves, splashes and foam, accurate viscous liquids, gases, fire, smoke, and explosions. It provides a high degree of control while preserving realistic fluid motion, this comes from the underlying algorithms that drive the simulations in the program but also from the new GUI design. The program itself is only a dynamics solver and does not support rendering, however it does produce extremely realistic water and fluid simulations with a nodal-based user interface.

Naiad's interface allows for the same flexibility and functionality that Houdini's interface does. The user creates nodes and then connects them together to form a relationship. Each node can be thought of as a function that will be executed on the incoming data. The incoming data is represented by links or connections between the nodes with lines. A description of a fluid simulation is expressed in a nodal-based graph that shows the user exactly how everything in the scene was created. The Naiad interface has been so successful that other companies have approached Exotic Matter about licensing the GUI for completely different products and applications [33].

CHAPTER III

METHODOLOGY AND IMPLEMENTATION

III.1. PyQt and Qt Designer

Out of the two scripting languages available for me to use with Maya (Mel and Python), I chose Python because it is cross-platform, its ease of use, and its support for Qt (PyQt). As far as interface design goes, MEL offers a limited selection of standard widgets such as combo boxes, radio buttons, and check boxes but does not offer the robust 2D graphic capabilities that PyQt has. These 2D graphics were essential in making my interface functional because I knew that I was going to need the capability to connect graphical nodes together with lines.

I chose PyQt because the developers of Qt have created a program called Qt Designer that allows an interface to be interactively built from Qt components. Qt Designer offers the ability to create widgets or dialogs in a what-you-see-is-what-you-get (WYSIWYG) manner and can be created, customized, and tested using different styles and resolutions [2]. This allows quick interface setup by dragging and dropping different widgets into a work area. One great benefit of using Qt Designer is that if the design of the interface is changed, only the interface file needs to be recompiled, which is done with a simple one-line command in a terminal (UNIX, Mac) or a command window (Windows).

Qt Designer was used extensively throughout this GUI to setup the base interface. Additionally, it was used to create the base of every widget menu that is displayed when a user clicks on a node in the scene. Even though Qt Designer has the ability to create signals and connect them to slots, I found it easier to program these slots and signals manually in the code–mainly for clarity. The rest of the interface's

functionality was coded and tested by hand without the help of Qt Designer.

III.2. Design of the Main Window

The inspiration for the look of the main window came from Autodesk 3ds Max. User's create particle systems and control their behaviors within event windows with a separate interface within 3ds Max called Particle Flow (PFlow). See Fig. 6. Even as a beginner it was easy to setup a particle system and control its behavior with PFlow's node-based interface.

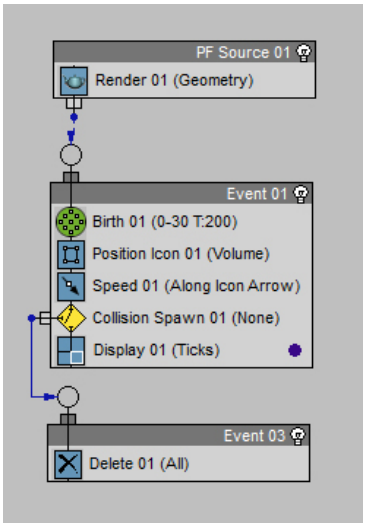


Fig. 6. Autodesk 3ds Max PFlow Events.

Although I did not recreate PFlow's event driven approach to creating particles, I did use the organization of the interface to help guide the organization of mine. PFlow's GUI and my GUI are organized into four separate frames that represent different "chunks" of data: the list of nodes, the description of the node, the menu for the node, and an area where a node can be dragged and dropped. See Fig. 7 and Fig. 8.

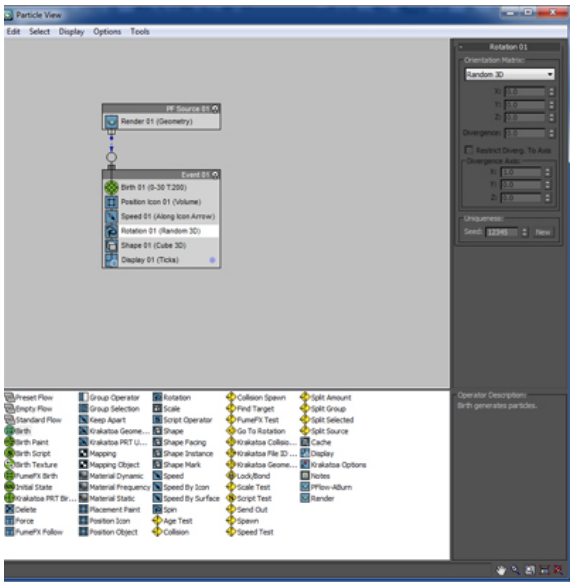


Fig. 7. Example of Autodesk 3ds Max's PFlow Interface.

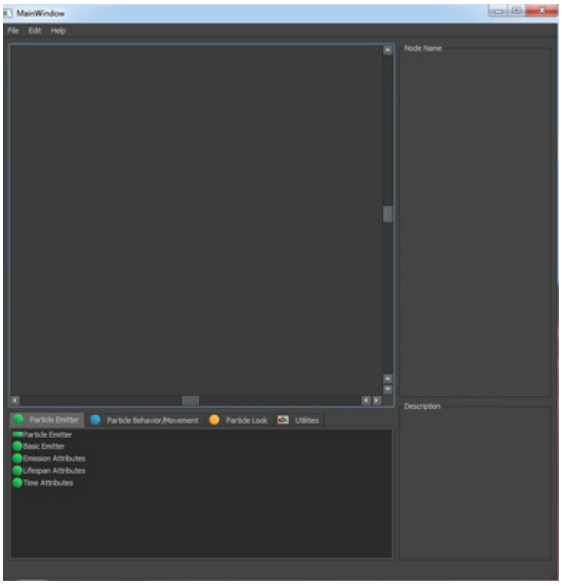


Fig. 8. Example of My Interface.

PFlow was a good guide to use for my interface because it follows ObjectSoft's (D. Sarna and G. Febish [32]) three design principles for good GUI design to arrange the screen into chunks of data, group chunks within frames, and line chunks up neatly into rows or columns.

The description of the nodes and the node menu portion of the interface are self-explanatory. When a user clicks on a node in the list of nodes, a couple of helpful sentences explaining what that node is used for will pop up in the "Node Description" frame. When a user clicks on a node that is in the scene, a unique menu will be shown in the "Node Menu" frame with attributes that the user can modify to affect the particle system.

My interface's list of nodes is a slightly modified version of PFlow's. In PFlow, nodes are organized alphabetically and also by color . The user is not informed of what the colors mean, but can infer that each color represents a different category of behaviors–some affect movement, some create particles, etc However, in my interface, I decided to group the nodes together by proximity–into separate tabbed windows–as well as by color similarity–two of the six principles of good graphic design laid out by A. Marcus [17]. By clinging to these two principles, a nodes meaning should be conveyed to the user more clearly as well as help in the finding of desired nodes more quickly.

In my interface, users are able to drag a node from the list of nodes and drop it into the scene with a technique known as serialization and deserialization–Python's version of serialization and deserialization is called pickling and unpickling. According to Microsoft [19], "Serialization is the process of converting the state of an object into a form that can be persisted or transported. The complement of serialization is deserialization, which converts a stream into an object. Together, these processes allow data to be easily stored and transferred." Serialization is necessary because a

node can exist in two forms: as a list object and as an icon-based graphic in the scene. When a list node starts to be dragged by the user, its information is pickled. When that node is dropped into the scene, the information is then unpickled and that list node is then converted to its icon-based form.

Once a node is in the scene, the user can then connect it to other nodes of the same color. A line and an arrow that always points to the input of the next node represent the connection between nodes . The arrow indicates which nodes are affecting other nodes. This portion of my interface is what differs the most from PFlow. In PFlow, connections are created between "event windows" and not the nodes themselves. These event windows house the nodes that affect the particle system and then the functions of the nodes are executed in a top to bottom order. E.g. Whatever node is at the top of the list in an event window will be executed followed by the second, etc Houdini's approach to connecting nodes is what I follow more closely–a node can take a multitude of input connections (depending on the node), a function attached to that node is then executed on the incoming data, and then the node will have that data as it's output.

III.3. Design of the Nodes

III.3.1. Organizing the Nodes

Maya assembles particle attributes into different groups that are poorly organized and scattered between two interface menu tabs: emitter and particleShape. See Fig. 9 and Fig. 10. The emitter tab is fairly straightforward. It contains attributes that relate to a particle being emitted, such as the emission speed, the emitter type, the rate of emission, etc The particleShape tab, however, is not as straightforward. This tab contains attributes related to velocity, acceleration, render type, render settings,

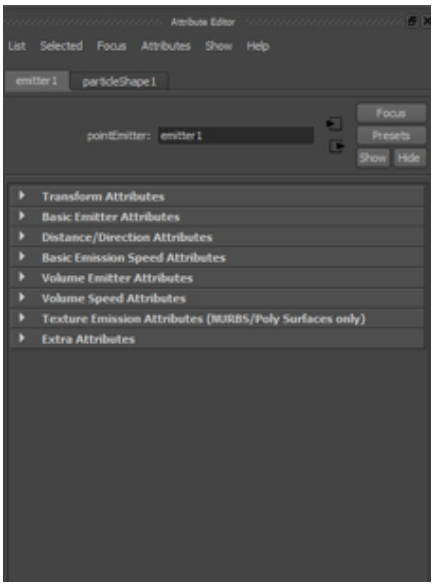


Fig. 9. Autodesk Maya 2011 Emitter Tab.

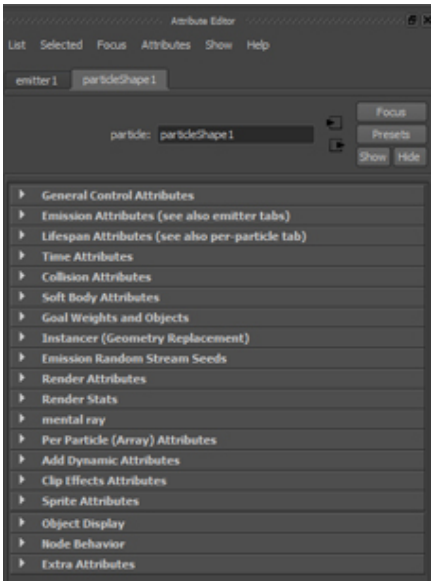


Fig. 10. Autodesk Maya 2011 particleShape Tab.

and even contains attributes that relate to particle emission (Why this is not under the emitter tab is not known).

This lack of organization in Maya's menus can confuse and frustrate users leading to less productive and efficient use of the program. One of the first tasks required in simplifying and better organizing these menus was to break down the particle system attributes into different categories. I discovered three different categories that all attributes could fit into: Particle Emitter, Particle Behavior, and Particle Look.

The Particle Emitter category houses all the attributes that deal with a particle being emitted. This includes: lifespan, the rate at which particles are emitted, whether or not the particle is being emitted from an object, etc The Behavior category contains all the attributes that deal with how the particle moves. This includes: forces that would act on the particle (Gravity, Wind, etc), its velocity after birth, collision objects, etc And finally, the Look category represents all the different ways a particle can be rendered: multistreak, points, spheres, other objects (known as instancing), etc

After discovering these categories, I decided to use this as the basis for my node setup and create two different node types for my interface: category and attribute. The three category nodes would be: Particle Emitter, Particle Behavior/Movement, and Particle Look. Each category node would have its own menu in the interface that would contain a list of the different attributes that related to that category. Color was then assigned to the different categories.

Following the advice of Robertson, Murch, Durrett and Trezona, I limit the interface to use a small color palette: green would be used to color all attributes that relate to a particle being emitted; blue would be used to color all attributes that relate to a particle's movement and behavior; orange would be used color all attributes that relate to a particle's look–originally the particle look category was colored red (for

a simple RGB color scheme), however, most people would identify something being colored red because of an error; and yellow would be used as the "selection" color for when a user selects any one of the nodes or its connecting line.

III.3.2. The Nodes

As powerful as Houdini and Naiad's interfaces are, the nodes themselves are fairly simple and share four traits: a base graphic to represent the node, a name that can be edited by the user, connection points, and a menu of attributes. See Fig. 11. Since the functionality of Naiad and Houdini's interface most closely resembles the functionality of my interface, I relied on their graphical representation of a node to guide the design of the category and attribute nodes in my GUI.

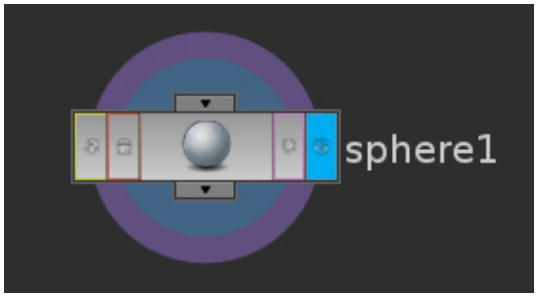


Fig. 11. Example of a Node in Sidefx Houdini.

III.3.2.1. Category Node

The category node is represented by a large colored rectangle–the color represents whatever color was assigned to that category. Both the particle behavior/movement category node and the particle look category node have two connection points, where as the particle emitter category node has three–this will be explained in the following

section. The node also has a text label that the user is free to edit. See Fig. 12.

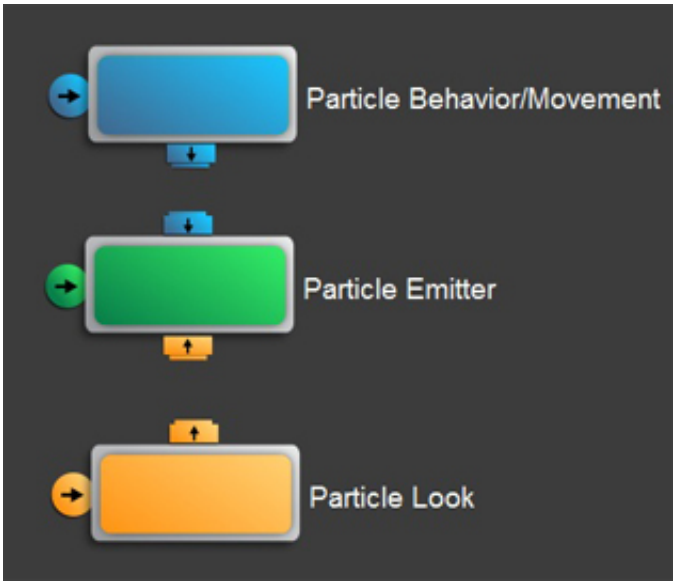


Fig. 12. Example of the Category Nodes in My Interface.

III.3.2.2. Attribute Node

The attribute node is similar to the category node in function but not shape. The attribute node is a round shape that is graphically smaller than the category node. Additionally, all attribute nodes have two connections as well as an editable text label. See Fig. 13.

III.3.2.3. Utility Node

A node not mentioned before is the utility node. The utility node is in its own category because it does not directly modify a particle system; it is meant to be used as a node that can connect to and modify an attribute node. The utility node has one connection point, a unique graphic, and an editable text label. See Fig. 14.

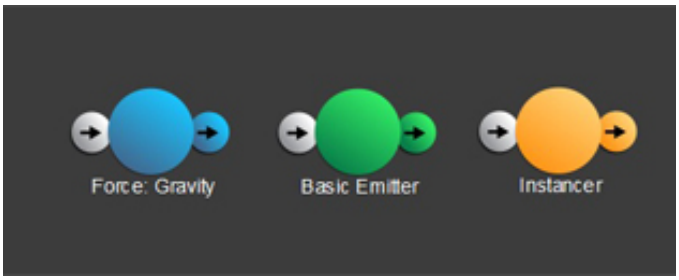


Fig. 13. Example of the Attribute Nodes in My Interface.

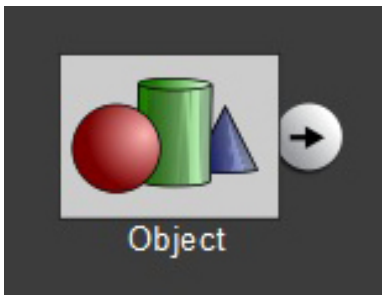


Fig. 14. Example of the Utility Node in My Interface.

All input and output connection points on every node have graphical arrows either pointing to the node itself or out of the node, which indicates whether or not that connection point is an input or an output. Further, when a user creates a connection line, the line itself has an arrow that always points from the output of one node to the input of the node it is sending data to, indicating the flow of data.

III.3.3. Node Widget Menus

The way that my interface interacts with Maya is through the widget menus, or the menu of attributes, that are assigned to each node. Each one of these menus are unique and house the different options that the user can modify to manipulate their

particle system. The menus can either be created in Qt Designer or completely from scratch in PyQt. Once the menus are created, the different widgets in the menu are given functions to execute. If the widget should be changing a value or doing some routine in Maya, PyCmds–the way that Python executes Maya commands–would be used.

These widget menus are really the heart of the interface. The functionality of the different menus are designed to alleviate some frustrating aspects of Maya's interface. For example, to create a dynamic relationship between objects in Maya, the user has to click on one object, shift+click on another object, and then select the dynamic relationship function in one of the menus. The problem with this method is that the user does not know which order he or she should click objects. It is not until after the objects are selected and the function is executed until the user knows whether or not the objects were picked in the correct order. If the objects were in the correct order then everything works fine, however, if the object selection was in the wrong order, an error is brought up and the user must repeat the process. This may not be a big inconvenience for the first few times, but on a large scale project where many dynamic relationships must be created, getting this error repeatedly would be frustrating and a time waster.

III.4. The Node Diagram

After deciding on the types of nodes I would have in my interface and how I would organize them, the node diagram had to be designed. The node diagram can be thought of as a flow chart that shows the user the flow of data between connected nodes. A connected line between each node (usually accompanied by an arrow on the line pointing to the node that the data is being sent to) indicates the flow of data.

In most 3D programs with a node-based interface, the flow of data is in a top-to-bottom and/or left-to-right format . Some programs, such as Houdini and Naiad, are primarily top-to-bottom, whereas Eyeon's Fusion, a node-based compositing program, is left-to-right and top-to-bottom .

Initially, I wanted to create a completely top-to-bottom node diagram. In this system, the user would create a "Particle Emitter" node as the top-most node and then connect that to any attribute node underneath it. The problem with this, though, was that my system is made up of three different colored categories of nodes and being able to have multiple colored nodes connected together looked confusing and disorganized ; this same cluttered look also applied to a left-to-right node diagram.

To alleviate this cluttered, disorganized look, I decided to use the category nodes as the connection hub for all of their attribute nodes to connect to. Additionally, the particle emitter category node would be used as the connection point for the other two category nodes: behavior and look. The particle emitter category node would now be the central hub for all connections. See Fig. 15. This new structure lead to a more organized and clean looking interface as well as being an indicator of how nodes should be connected. The new organized look resembled a tree like structure (where the category nodes are the trunk and the attribute nodes are the branches) and evolved into a left-to-right flowing diagram. Since color was going to be a vital part of showing the user how the interface worked, careful consideration had to be given to the parts of the node that would be colored.

The utility node is unique in that there is only one of them in my interface, called an "Object Utility", and can be connected to nodes from all three categories. The "Object Utility" node is used to house information of a single object in a scene. Because of this, the main icon graphic is of three basic 3D shapes (giving the user instant visual recognition of what the node is). It has one circular output connection

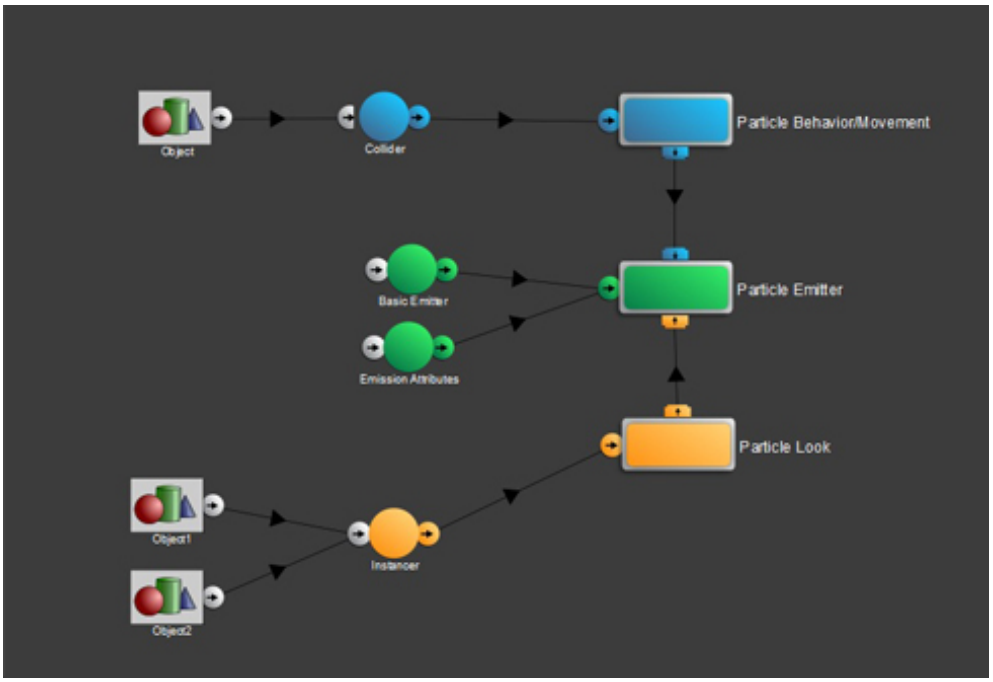


Fig. 15. Example of the Node Graph in My Interface.

point and it is colored silver.

The attribute node's big round shape is colored whatever color is appropriate for it and has an input connection point and an output connection point. E.g. if the attribute node is from the behavior category, it is colored blue. The input connection point is circular and is colored silver. The reason why it is silver is to show that the output of a utility node (circular and silver) could be connected to the input of the attribute connection. The output connection is circular and is colored whatever is appropriate for its type.

The particle and behavior category nodes are similar in functionality and look, the only difference being their color. Each of those nodes have an input connection graphic that matches the output connection on an attribute note (circular and of

the same color), indicating that they can be connected. Also, the look and behavior category nodes have one output connection. The output connection is a colored rectangular shape.

Finally, since the behavior and look nodes needed to connect to a particle emitter category node, I felt that the emitter node should be, literally, in the center of the behavior and look nodes (since it was being used as a central hub for all connections). Like the other two category nodes, the emitter node has a circular input connection for its attribute nodes, however, different than the behavior/look node, the emitter node actually has two other input connections located on the top and bottom of the node and no output connections. The top input connection matches the output connection of the behavior category node and is rectangular and blue. The bottom input connection matches the output connection of the look category node and is rectangular and orange.

CHAPTER IV

RESULTS

There are a total of twenty-five nodes in the interface. Each node was given a unique widget menu that was initially developed in Qt Designer. After the widget menu was created, functionality with Maya was then added by using PyCmds. All nodes are separated into four different categories: Particle Emitter, Particle Behavior/Movement, Particle Look, and Utilities. Each category is displayed in its own individual tabbed menu in the GUI and is also given a unique color to distinguish it from the other categories. Because the attribute nodes only function if they are connected to a category node, each category node is displayed at the top of the tabbed menu; the rest of the attribute nodes are in alphabetical order so the user can find desired nodes faster.

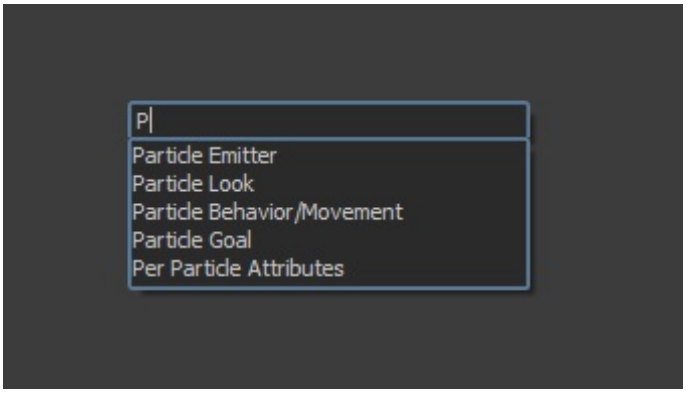


Fig. 16. Example of the Context Menu in My Interface.

Also added in the GUI is tabbed context menu–something present in almost all node-based interface designs. This menu is shown when the user presses the Tab key in the node scene. The menu allows users to type in the name of the node and then press Enter to lay that specific node down in the scene. See Fig. 16. What is unique

about this menu is that as the user is typing in the name of a node, only the nodes that match with the typed name will be displayed in the menu. This allows users to greatly speed up their use of the interface by only having to press a few keys on the keyboard instead of dragging and dropping every node into the scene. Of course, this menu does not have to be used, but it is an option available for users that may be coming from other programs that have a tabbed context menu.

IV.1. Particle Emitter Category

The Particle Emitter category can be related to the "emitter" tab in the Maya interface (found in a particle system) and contains six different nodes: Particle Emitter, Basic Emitter, Emission Attributes, Lifespan Attributes, Object Emitter, and Time Attributes. See Fig. 17. The Lifespan Attributes and Time Attributes were moved into this category and out of the "particleShape" tab in Maya because these options are set when a particle is birthed. All of these nodes have all the basic functionality that is present in Maya.

Fig. 17. The Emitter Tab.

The Particle Emitter category node is what the user must create if he or she

wants a particle system to be created. As soon as this node is created in the scene, a particle system is created in Maya and selected. The widget menu for this node shows the name, the translation, and the rotation of the particle system (as it was created in Maya). Fig. 18. The translate and rotate attributes in the widget menu will update if the user moves or changes the translation or rotation attributes in Maya.

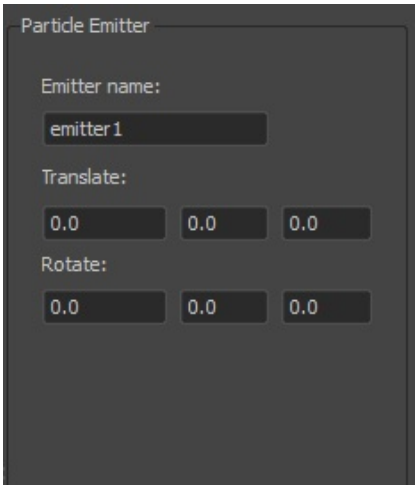


Fig. 18. Particle Emitter Category Widget Menu.

The Object Emitter node is unique and allows users to connect an Object Utility node into it to create an object emitter. This allows users to clearly see what object is being used as a particle emitter and eliminates the need for a user to use, the clunky, click an object and shift+click a particle system (or click a particle system and shift+click an object).

IV.2. Particle Movement/Behavior Category

The Particle Behavior/Movement category contains fourteen different nodes and is where most of the nodes in the interface are located. See Fig. 19. Five of the

nodes are default options found in a particle system in Maya: Basic Emission Speed, Distance/Direction Attribute, General Control, Per Particle Attributes, and Volume Speed Attributes. The other eight nodes deal with forces and collision objects that act on a particle system as well as events that occur when a particle collides with an object–emitting new particles for instance.

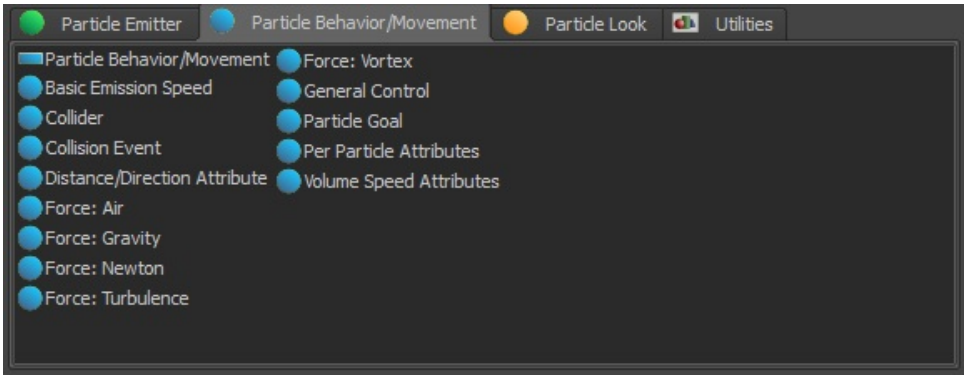


Fig. 19. The Behavior Tab.

The menu for the Per Particle Attributes node is a modified version of the default Per Particle Attributes menu in Maya. The default PP Attributes menu is shown in Fig. 20 and is worked by users right-clicking an empty line and either choosing Creation Expression, Runtime Expression Before Dynamics, or Runtime Expression After Dynamics. A separate window pops up where an expression can be typed in. One may think that this window is unique to each attribute listed (Position, Velocity, Acceleration, etc...), however, this is not the case; the window that pops up is shared between all the attributes. In my PP Attribute menu, I make this more clear to get rid of any confusion a user might have.

The new menu is shown in Fig. 21 and has three different combo box options to evaluate the expression At Creation, Before Dynamics, and After Dynamics. Each

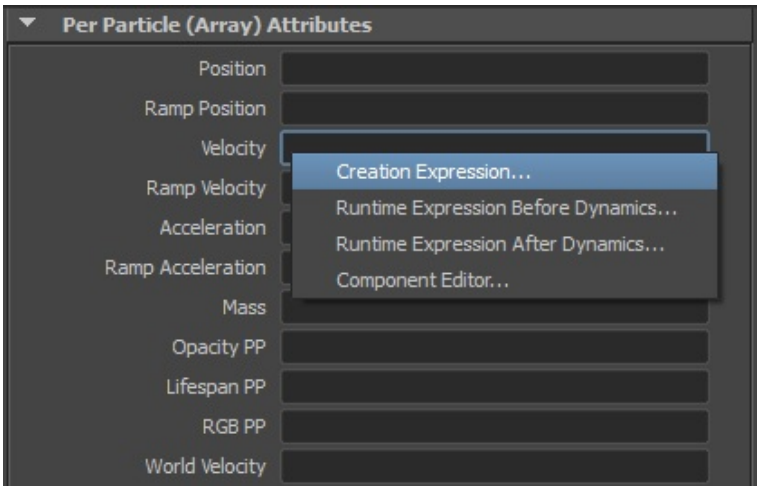


Fig. 20. The Default Per Particle Attribute Menu in Autodesk Maya 2011.

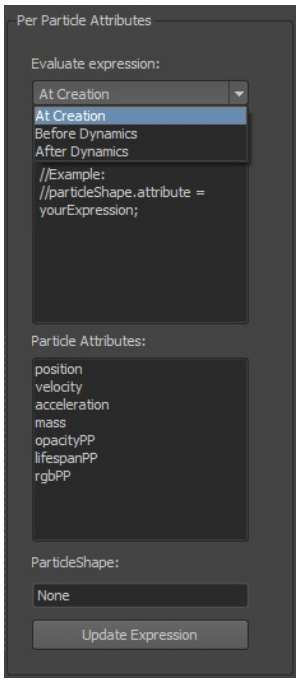


Fig. 21. The Per Particle Attribute Menu in My Interface.

one of these options has a unique area for editing an expression, which also shows a very concise example of how to format an expression. Below the expression editor is a list of the different particle attributes that a user can affect with an expression. This menu can be used as a reference, or the particle attributes can be drag and dropped into the text editor to create the correct text. E.g., If "position" is dragged and dropped into the text editor, "particleShape.position" will be appended to whatever text is in the editor. Finally there is an "Update Expression" button that actually creates/updates the expression in Maya; as opposed to the "Edit" button in Maya that serves as the same function as my "Update Expression" button but with a confusing name.

Creating a force or collision object in my interface is different than in Maya. The original way of creating a force that affects a particle system in Maya is to go to the Fields menu, click the field to create, click the field, use the clunky shift+click method to click the particle system, again go to the Fields menu, and select "Use Selected as Source of Field." This process takes seven clicks of the mouse. If the user did not shift+click the particle system and the field in the correct order, the user can expect to add four more clicks for a total of eleven clicks. In my system if a user wants a force to affect the particle system, all that has to be done is to connect a force node to a blue category node–a total of three clicks. Once this connection is made, the node tells Maya to create one of five fources (air, gravity, newton, turbulence, vortex) and then automatically create a dynamic relationship between the particle system and that force. To create a collision object in my interface, a "Collider" node must be laid down as well as an "Object Utility" node. After the user picks an object with the "Object Utility" node, it can then be connected to the "Collider" node. The node then automatically sets up the dynamic relationship in Maya–for a total of five mouse clicks. Setting up a collision object in Maya the default way is

a total of five clicks. However, if the user gets the shift+click of the collision object and the particle system wrong, it adds another three clicks. Those three clicks aren't much, but they add up over time along with frustration with using the shift+click method. Having a user connect a line to make a relationship between objects (instead of the shift+click method) is a simpler and less frustrating way to work–there is no possibility of selecting objects out of order.

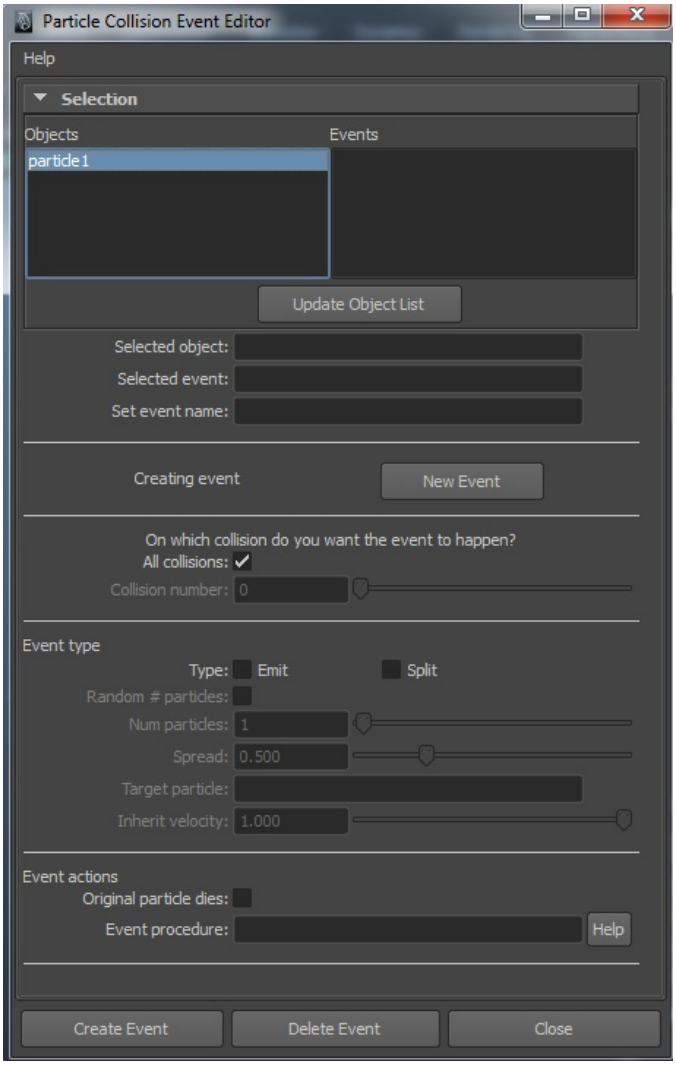


Fig. 22. Default Particle Collision Event Editor in Autodesk Maya 2011.

The “Collision Event” node was one of the more complex nodes to create because the amount of options available to change. This node modifies the “Particle Collision Event Editor” menu in Maya and is shown in Fig. 22. For a new event to be created, a user must select the particle name from the top left list, then press the “Create Event” button. However, the key item in this menu is the “Target particle” text editing line. This line determines what particle system the new particles will take attributes from. By default, if an event is created, the particle system that the new particles take attributes from is itself. Though, the user can type in a different existing particle system if he or she desires. The problem with this is that there is no immediate indication or notice that the user can pick a different particle system. In my system, when a user connects a “Collision Event” node into a blue category node, the first thing that happens is a separate dialog asking whether or not the user wants the resulting particles to be created by a new particle system or the current one. See Fig. 23.

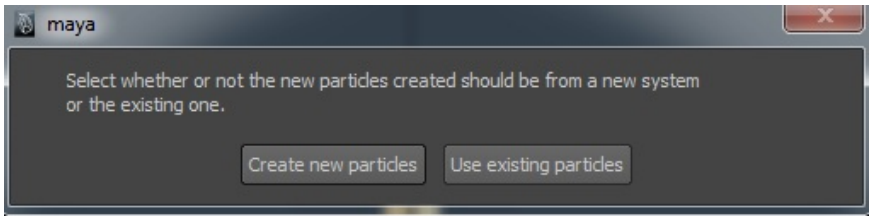


Fig. 23. Resulting Dialog Box from a User Laying Down a Collision Event Node.

If the option for a new particle system is selected, a new “Particle Emitter” node is created in the scene with a connected dotted line to the “Collision Event” node. The dotted line is an indication to the user that there is a relationship between the two nodes. After an option is picked, the node automatically sets up the collision event and the user is free to change any attribute in the menu. All of this can be

setup with about two clicks of the mouse. In Maya, the default way of creating a collision event is about five or six clicks of the mouse *if* the user knows exactly which order to select the buttons in. The poor organization of the menu can easily cause confusion and induce misclicks in setting up a collision event.

IV.3. Particle Look Category

Fig. 24. The Look Tab.

The Particle Look category houses three different attribute nodes: Instancer, Render Attributes, and Render Stats. See Fig. 24. The "Render Stats" node has the same options as the default menu in Maya. The "Render Attributes" and the "Instancer" nodes in my interface are simplified and more organized versions of the default menus that are found in Maya.

The "Render Attributes" menu in Maya allows users to pick how their particles will render: MultiPoint, MultiStreak, Numeric, Points, Spheres, Sprites, etc... To work this menu, the particle render type has to be selected, then the "Current Render Type" button must be clicked on to add attributes for that render type . Initially this setup is fine, however, the problem with this menu is when multiple attributes are

added for multiple different render types. One would think that only the attributes for the selected render type would be shown, however, this is not the case.

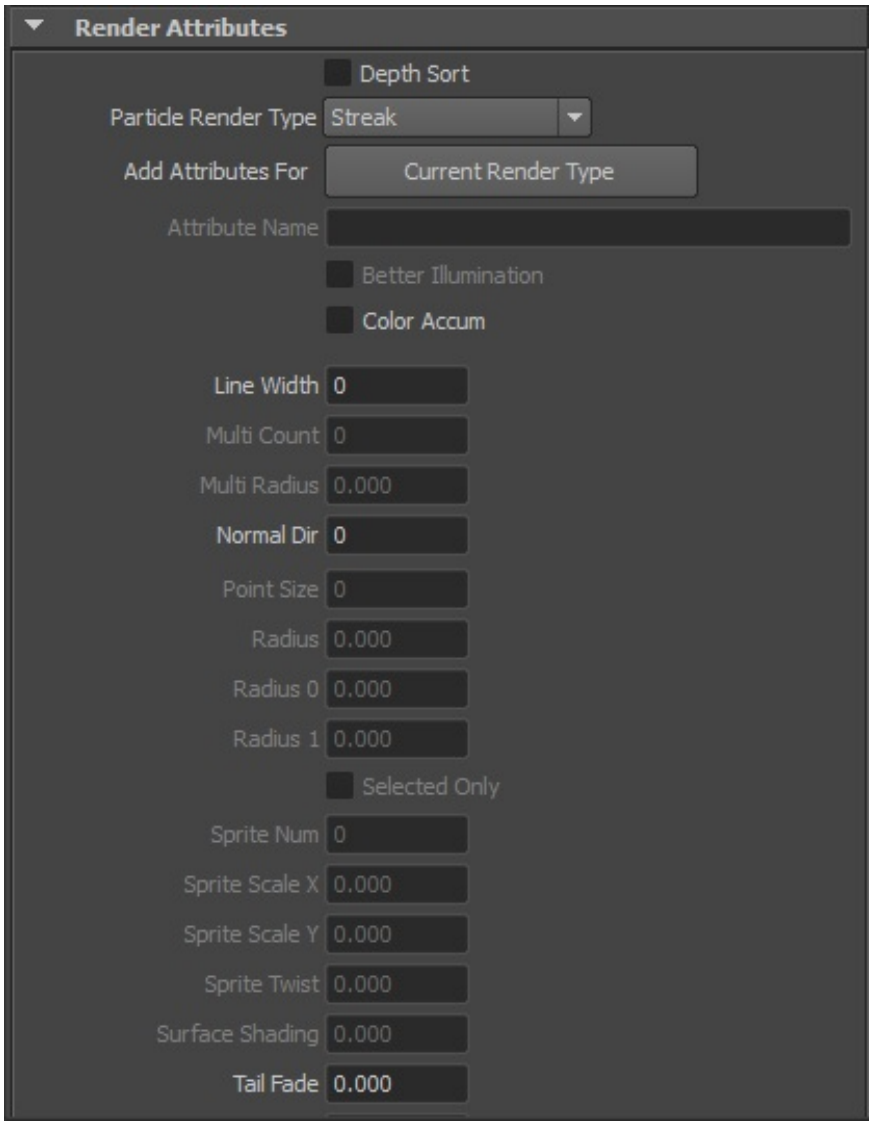
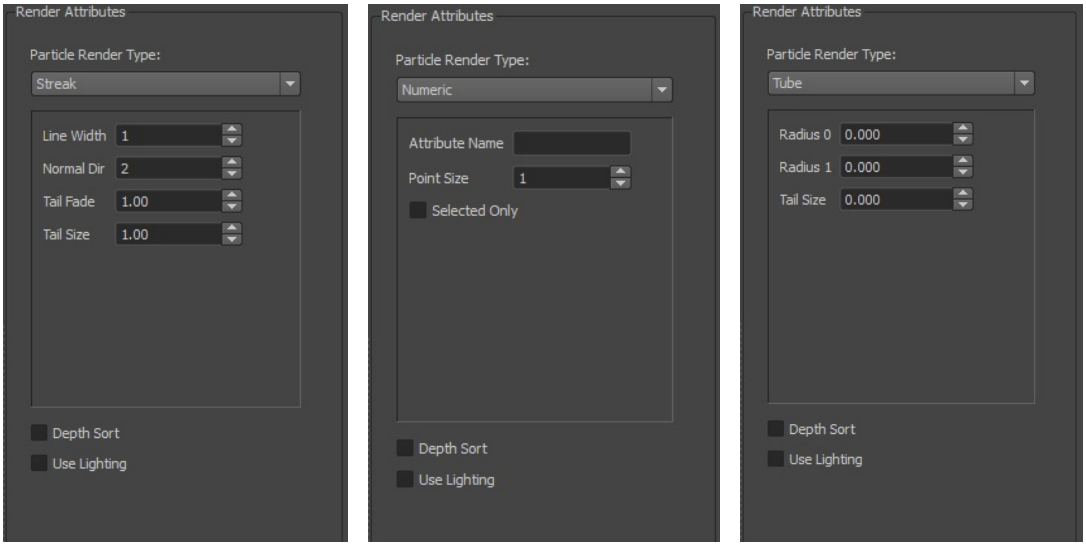


Fig. 25. The Default Render Attributes Menu in Autodesk Maya 2011.

Fig. 25 shows the "Render Attributes" menu in Maya after attributes for all render types have been added. What can be seen in the list are all the attributes for

every render type. The problem with this is that the attributes that can be changed for the selected render type are scattered throughout a multitude of different options and are not organized in any discernable way. To combat this unorganization, the "Render Attributes" menu in my interface only shows the attributes that are available for that selected render type. See Fig. 26.



(a) Streak (b) Numeric (c) Tube

Fig. 26. Examples of the Render Attributes Menu in My Interface: (a) The Streak Menu. (b) The Numeric Menu. (c) The Tube Menu.

Attaching a geometric object to a particle in a particle system is referred to as instancing. When a particle system is created in Maya, one of the options under the particleShape tab is called "Instancer (Geometry Replacement)". However, initially this menu is grayed out, there is no way to create an instanced object from this menu, and there is no immediate indication of what must be done to create instanced objects. See Fig. 27. To create an instanced object, the user has to go to a separate part of the interface and then select an option called "Instancer (Replacement)". This option then creates an "instancer" object (not in the geometric shape sense) that houses a

menu where objects can be added to an "Instanced Objects" list and attribute can be changed. The system is confusing and requires users to go between two separate menus to manage any instanced objects.

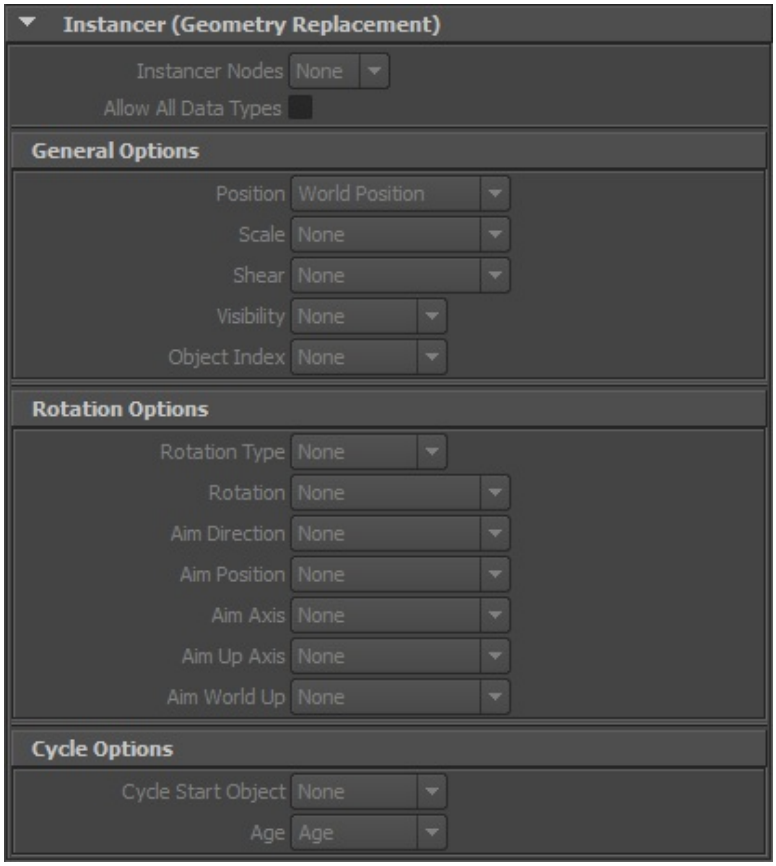


Fig. 27. The Default Instancer (Geometry Replacement) Menu in Autodesk Maya 2011.

My "Instancer" node simplifies this process a great deal. The new procedure is to connect at least one "Object" utility to an "Instancer" node. An arbitrary amount of "Object" utility nodes can be connected to the "Instancer" node, which will result in more objects being instanced onto the particles in the particle system. Also, all of the different options available for changing the behavior of instanced objects are

located in one menu. See Fig. 28. Not only does this new node simplify the creation of instanced objects, it also alleviates the need to click between two different menus.

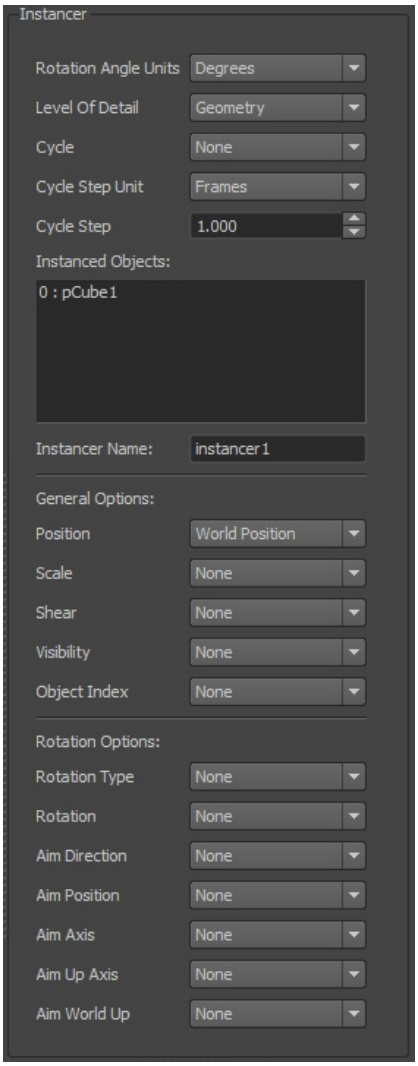


Fig. 28. The Instancer Menu in My Interface.

IV.4. Utilities Category

The Utilities category only has one node, called the "Object" node. With this node, objects can be selected within the Maya scene. This is clearly indicated by the large button entitled "Pick Object". Once this button is clicked, a text label is shown that says, "Awaiting selection from Maya..." After an object is picked from the scene in Maya, the node's menu is then populated with the different attributes of that object. Fig. 29 shows the node's menu after an object in Maya has been picked. All of the attributes in this menu can be changed, which will be reflected within Maya. Also, if the user changes the name, translation, rotation, or scale of the object in Maya, these attributes will be updated in my interface. Even though this node is fairly simplistic, it is a key element in making some of the nodes mentioned above work properly.

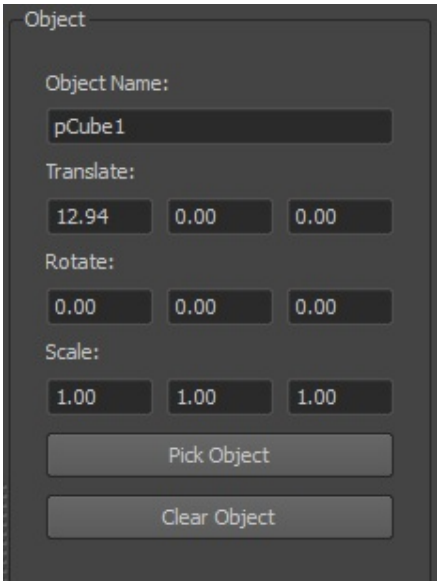


Fig. 29. The Object Category Node's Widget Menu.

CHAPTER V

CONCLUSION AND FUTURE WORK

In the end, a fully functioning node-based GUI was created for creating and modifying the behavior and look of particle systems in Maya. This interface uses inspiration from current programs that employ a node-based interface and that pays close attention to the color used throughout the interface as well as the organization of the data. Knowing objects that are located near each other will appear to belong together and objects that share a similar characteristic (e.g., color, size, shape) will be perceived as belonging together allowed me to understand why following simple design rules, such as, organizing data into logical chunks and grouping those chunks of data within frames, creates a more efficient and intuitive interface for users.

Each node's functionality was designed to alleviate possible frustrations in Maya. For example, the notorious shift+click method of creating dynamic relationships between objects. Furthermore, each node menu was created from scratch to be better labeled, organized, and have at least the default functionality of the menus in Maya. In a few cases, more functionality was added to menus so that users would only have to look in one area for attributes. With this new interface, users are able to create and setup particle systems more efficiently and intuitively. Also, users are able to visualize their particle system in a 2D graphical way as well be able to see exactly how that particular particle system was created.

This node-based interface can be further developed in a couple of different ways. First, to get more defined particle behavior in Maya, expressions must be written in for different particle attributes (velocity, acceleration, etc...). This requires knowledge of scripting and can sometimes require the knowledge of vector math. This can be

limiting to users that are new to 3D graphics or new users to Maya that may not be familiar with MEL syntax. Programs like SideFX Houdini attempts to eliminate the need to know script formatting by allowing users to use a visual programming language called Vex OPerators (or VOPS). VEX is a language in Houdini that is used to process large amounts of data and stands for Vector EXpressions. VEX is syntactically similar to the C programming language, although, if a user has never programmed in C before, this could be challenging. VOPS in Houdini is a node-based approach to building VEX operations and shaders. With VOPS, users connect vectors and vector functions graphically. See Fig. 30. This type of node-based programming approach could be implemented in my interface to be used with the "Per Particle Attribute" node to allow users to achieve a greater level of control over their particle system without having intimate scripting knowledge of MEL.

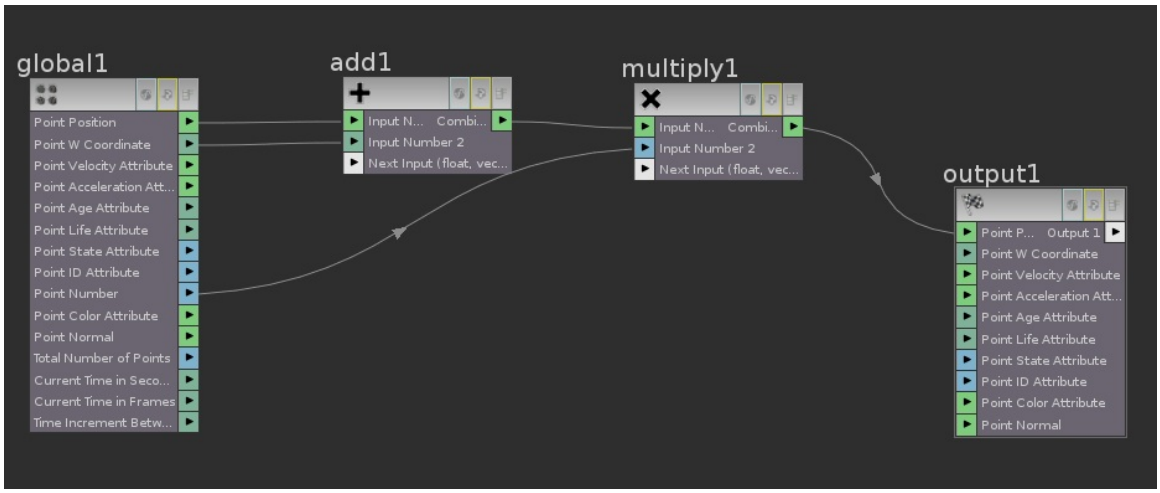


Fig. 30. A Simple VOPS Graph in SideFX Houdini.

Second, this node-based interface can be expanded into different areas of Maya's interface. I believe that some aspects of Maya would not benefit very much from a

node-based interface, such as, animation, lighting, or rendering. Texturing in Maya is done through the hypershade window is already node-based, though; the graphical portion of this could be improved greatly. Also, modular rigging has potential to benefit from a node-based interface. In fact, a current student here at Texas A&M University has approached me with plans (and my permission) to use this interface for his modular rigging based thesis.

Finally, this interface could be opened up to public critiquing. Since the GUI is developed with Python and PyQt, both of which are open-source and free to use for the public, people would be able to download and test this GUI (as long as they have access to Maya 2011). Allowing Maya users and non-Maya users with different backgrounds to use the interface and offer their ideas and suggestions of how to improve it would greatly benefit the use of the interface as well as it sparking ideas of new nodes and functionality to add.

REFERENCES

[1] "Procedural workflow," http://www.sidefx.com/index.php?option=com_content&task=view&id=1587&Itemid=333, September 2011.

[2] "Qt designer manual," http://developer.qt.nokia.com/doc/qt-4.8/designer-manual.html, October 2011.

[3] "Iterative virtual prototyping: Linking houdini with radiance and energyplus," March 2012. [Online]. Available: http://wiki.epfl.ch/loveridge/documents/70.pdf

[4] M. Boshernitsan and M. Downes, *Visual programming languages: A survey*. Citeseer, 2004.

[5] M. Burnett and M. Baker, "A classification system for visual programming languages," *Journal of Visual Languages and Computing*, vol. 5, no. 3, pp. 287–300, 1994.

[6] S. Chang, "Visual languages: A tutorial and survey," *Visualization in Programming*, pp. 1–23, 1987.

[7] M. Chignell and J. Waterworth, "Wimps and nerds: an extended view of the user interface," *ACM SIGCHI Bulletin*, vol. 23, no. 2, pp. 15–21, 1991.

[8] R. Christ, "Review and analysis of color coding research for visual displays," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 17, no. 6, pp. 542–570, 1975.

[9] M. Clasen. (2012, March) Gtk+ history. [Online]. Available: http://people.redhat.com/mclasen/Usenix04/notes/x29.html

[10] E. Dumbill and N. Bornstein, *Mono (Developer's Notebook).* O'Reilly Media, Inc., 2004.

[11] J. Durrett and J. Trezona, "How to use color displays effectively," *Byte*, vol. 50, p. 53, 1982.

[12] J. Hadamard, *An essay on the psychology of invention in the mathematical field.* Dover Publications, 1954.

[13] B. Ives, "Graphical user interfaces for business information systems," *MIS Quarterly*, pp. 15–47, 1982.

[14] J. Janhager, "User consideration in early stages of product development: Theories and methods," Ph.D. dissertation, Department of Machine Design, Royal Institute of Technology [Institutionen för maskinkonstruktion, Kungl. Tekniska högskolan], 2005.

[15] P. Janssen and K. Chen, "Visual dataflow modeling: A comparison of three systems," *Designing together - CAADFutures 2011*, 2011.

[16] L. MacDonald, "Using color effectively in computer graphics," *Computer Graphics and Applications, IEEE*, vol. 19, no. 4, pp. 20–35, 1999.

[17] A. Marcus, "Computer-assisted chart making from the graphic designer's perspective," *ACM SIGGRAPH Computer Graphics*, vol. 14, no. 3, pp. 247–253, 1980.

[18] ——, "Designing graphical user interfaces," *UnixWorld (October 1990)*, pp. 135–138, 1990.

[19] Microsoft. (2012, March) Serialization. [Online]. Available: http://msdn.microsoft.com/en-us/library/7ay27kt9%28v=vs.80%29.aspx

[20] G. Miller, "The magical number seven, plus or minus two: some limits on our capacity for processing information." *Psychological review*, vol. 63, no. 2, p. 81, 1956.

[21] J. Miller, "Living systems: The organization," *Behavioral Science*, vol. 17, no. 1, pp. 1–182, 1972.

[22] D. Morris, "A new graphical user interface for a 3d topological mesh modeler," 2008. [Online]. Available: http://repository.tamu.edu//handle/1969.1/85977

[23] G. Murch, "The effective use of color: Physiological principles," *Tekniques*, vol. 7, no. 4, pp. 13–16, 1984.

[24] B. Myers, "A new model for handling input," *ACM Transactions on Information Systems (TOIS)*, vol. 8, no. 3, pp. 289–320, 1990.

[25] ——, "The state of the art in visual programming and program visualization," vol. CMU-CS-88-, 1988.

[26] D. Norman, "Some observations on mental models," *Mental Models*, vol. 7, no. 112, pp. 7–14, 1983.

[27] C. Pancake, "Principles of color use for software developers," *Tutorial M1 from supercomputing*, vol. 95, p. 90, 1995.

[28] A. Peslak, "A framework and implementation of user interface and human-computer interaction instruction," *Journal of Information Technology Education*, pp. 189–205.

[29] J. Reimer, "A history of the gui," *Ars File: Paedia from Ars Technica, LLC (http://arstechnica. com/articles/paedia/gui. ars)*, 2005.

[30] P. Robertson and I. B. M. Corporation, *A guide to using color on alphanumeric displays*. International Business Machines Corporation, 1980.

[31] D. Salber, A. Dey, and G. Abowd, "The context toolkit: aiding the development of context-enabled applications," *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, pp. 434–441, 1999.

[32] D. Sarna and G. Febish, "What makes a gui work?" *Datamation*, vol. 40, no. 14, p. 29, 1994.

[33] M. Seymour. (2010, April) The tech behind the tools of avatar part 2: Naiad. [Online]. Available: http://www.fxguide.com/featured/The_Tech_Behind_the_Tools_of_Avatar_Part_2_Naiad

[34] N. Shu, Ed., *Visual programming*. New York, NY, USA: Van Nostrand Reinhold Co., 1988.

[35] A. Sloman, "Interactions between philosophy and artificial intelligence: The role of intuition and non-logical reasoning in intelligence," *Artificial Intelligence*, vol. 2, no. 34, pp. 209 – 225, 1971. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370271900117

[36] D. Smith. (1975, January) Pygmalion: An executable electronic blackboard. [Online]. Available: http://acypher.com/wwid/Chapters/01Pygmalion.html

[37] M. Summerfield, *Rapid GUI Programming with Python and Qt*. Ann Arbor, MI: Prentice Hall, 2010.

[38] P. Wright, D. Mosser-Wooley, and B. Wooley, "Techniques and tools for using color in computer interface design," *Crossroads*, vol. 3, pp. 3 – 6, March 1997.

[39] T. Yuanhua, A. Osvalder, and M. Karlsson, "Considering the importance of user profiles in interface design," pp. 61–80, 2009. [Online]. Available: http://www.intechopen.com/books/user-interfaces/considering-the-importance-of-user-profiles-in-interface-design

VITA

Timothy Clayton Withers
c/o Joshua Bienko
College of Architecture
Texas A&M University
C108 Langford Center
3137 TAMU
College Station, Texas 77843-3137
clay.withers@gmail.com

Education

M.S., Visualization,	Texas A&M University, August 2012
B.S., Environmental Design,	Texas A&M University, May 2009